Practical - 01

Date:

Aim: - **Introduction to PROLOG.**

What is PROLOG?

- The name PROLOG was taken from the phase "Programming Logic".
- Prolog is unique in its ability to infer (derive by format reasoning) facts and conclusion from other facts.
- To enlist Prolog's help in solving a problem, the user attempts to describe the problem in a logically coherent and format manner, presenting facts and knowledge about the problems and specifying a goals.
- The computer uses this knowledge to achieve the specified goal, defining its own procedure.

Difference between Procedural language & Object-oriented language

Procedural Languages	Object - Oriented Languages
➢ BASIC, COBOL, Pascal	➢ Prolog, Lips
➢ Use previously defined procedure to solve problems.	➢ Use heuristics to solve problems
➢ Most efficient at numerical processing.	➢ Most efficient formal reasoning
➢ System created and maintained by programmer	➢ Systems developed and maintained by knowledge engineering.
➢ Use structured programming	➢ Interactive and cyclic development

APPLICATION FOR PROLOG

Expert System:-

- Expert systems are programs that use interface techniques that involve formal reasoning normally performed by human expert to solve problems on a specific area of knowledge.
- It can advice, diagnose, analyze and categories using a previously define knowledge

base.

- The knowledge base is collection of rules and facts often return in Prolog.
- The user is restricted to yes or no.

Natural Language Processing:-

- Natural language processors are part of expert system, permitting non-technical users to describe problems and resolve them.
- The user is not restricted to yes or no. The dialog is more natural and "human".
- It requires too much memory and processing power for today's microprocessor.

ROBOTICS:-

- Robotics is the branch of artificial intelligence concerned with enabling computer to see and manipulate objects in their environment.
- It is improved in studying and developing sensor system, manipulates controls and heuristic for solving objects and space oriented environment problem.
- Prolog facilitates the developments of robotics control problems that can use input data from sensors to control manipulate.

Gaming &Simulations:-

- Prolog is ideal for gaming and simulations involving formal reasoning.
- They employee a set of logical rules that control the play and action.
- It is possible to test various heuristic against a particular control strategy.

Features of Turbo Prolog:-

- You can compile stand-alone programs that will execute on a machine that is not running turbo prolog.
- Full complements of stand-alone predicates for many functions.
- Functional interface to other language is provided allowing procedural language support to be added to any prolog system.
- Declared variables are used to provide more secure development control.
- Both integer and real (floating point) arithmetical are provided.
- An integrated editor is provided. Making program development, compilation and debugging is very easy.

Limitations of PROLOG:-

- It does not support virtual memory.
- It is inefficient for numerical processing.
- Program size is only limited by disk space.
- The size of system developed with Prolog is limited by the amount of memory available.

Practical - 02

Date:

Aim: - **Study of Facts, Predicates, Clauses, Domain Types and Rules.**

FACTS

- Turbo Prolog permits to describe facts as symbolic relationship. For example, if the right spear in your stereo system is not emitting sound(is dead), you can express this in English as

  The right speaker is dead.

  This same fact can be expressed in Turbo Prolog as

  Is (right_speaker, dead).

- This factual expression in prolog is called CLAUSE. The period at the end of the clause.
- In this example right_speaker and dead are objects. An object is the name of the element of a certain type. It represents an entity or property of an entity in the real world.

PREDICATES

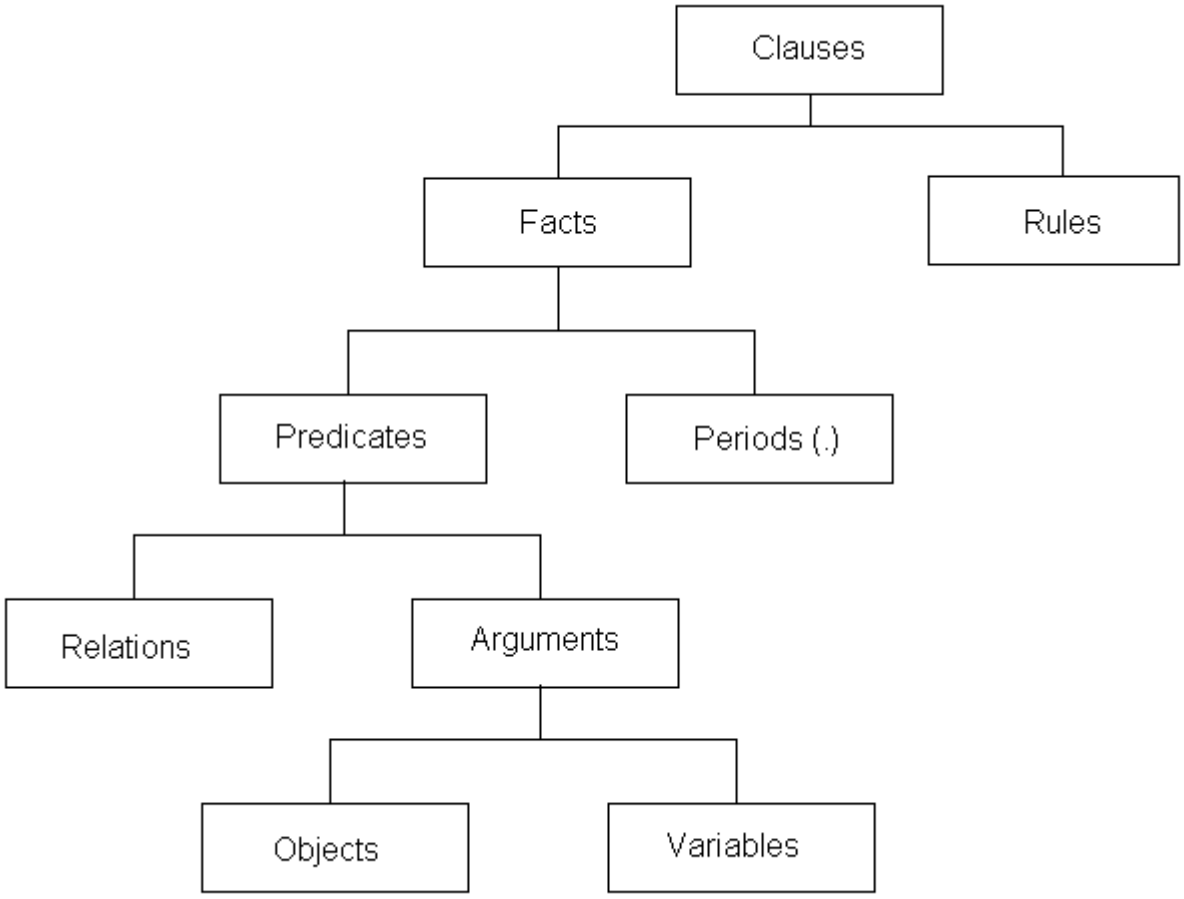
- The entire expression before the period in each case is called a predicate. A predicate is a function with a value of true or false.
- Predicate express a property or a relationship. The word before the parenthesis is the name of the relation.
- The element is the name of the relation. The arguments of the predicates, which may be object or variables. Here are a few examples of predicates,

  Employee(bill)

  Eligible(marry)

- If you add period to predicate, you have complete clause.

RELATIONSHIP OF CLAUSE, PREDICATES, RELATIONS, AND OBJECTS



CLAUSE

- A clause in prolog is a unit of information in a prolog program ending with a full stop.

  A clause may be a fact, like:

  Likes (marry, pizza)

- Or Rule like:

  Eats (person, Thing)

  Likes (person, Thing)

  Food (Thing).

CLAUSE RULE

- A given relation can be having any number of objects. A predicate can be having any number of arguments, including zero. The number of arguments called ARITY of predicate.
- An object name can be representing a name of entity (Bob, automobiles) or an abstract concept (defective). It can be noun, adverb, or adjective. It can be any sequence of characters that abides by the rule of the object type.
- Objects are always singular. In the last example above automobiles is considered a singular object.
- Certain names are reserved in turbo prolog and should not be used for the object names.
- The prolog expression does not have to contain all the words of the English expression.

DOMAIN TYPES

- Domain defines the types of each object. The domain section of the turbo prolog controls the typing of the object.
- Six basic object types are available to the user, char, integer, real, string, symbol and file.

Char **:** Single character enclosed between single quotation marks.

Integer : Integer from -32768 to 32767.

Real **:** Floating point number (1e-307 to 1e308)

String **:** Character sequence (enclosed between double quotation marks)

Symbol :Character sequence of letters, numbers and underscores, with the first character is a lower case letter.

File **:** Symbolic file name.

Example:-

In the case section

Is (right_speaker, defective)

In the predicate section

Is (component, status)

So, the Domain section contains,

Component status = symbol

RULES

- A rule is an expression that the truth of the particular facts depends upon one or more other facts.

  Example

  If there is body stiffness or pain in the joints

  AND there is sensitivity to infections,

  THEN there is probably a vitamin C deficiency.

  In the turbo prolog clause

  Hypothesis (vitc_deficiency)if

  Symptom (arthritis) and

  Symptom (infection_sensitivity).

- A rule expresses a relation between facts. Every rule has a conclusion (head) & an antecedent (body).
- As rules are important part of any prolog, a short hand has been developed for expressing rules. So, the previous rule becomes:

Hypothesis (vitc_deficiency):-

Symptom (arthritis),

Symptom (infection_sensitivity).

- The operator ":-" is called a break. A comma express an "and" relationship and semicolon (:) express an "or" relationship.
- Every rule has a conclusion (or head) and an antecedent (body). The antecedent consists of one or more premises.
- The premises in the antecedent from a conjunction of goals that must be satisfied for the conclusion to be true.
- If all the premises are true, the conclusion is true, if any premise fails, the conclusion fails.

Practical - 03

Date:

Aim: - **Write a program to demonstrate use and semantics of Facts.**

```
domains
        disease, indication= symbol
predicates
        symptom(disease, indication)
clauses
        symptom(chicken_pox, high_fever).
        symptom(chicken_pox, chills).
        symptom(flu, chills).
        symptom(cold, mild_bodyache).
        symptom(flu, sever_bodyache).
        symptom(cold, runny_nose).
        symptom(flu, runny_nose).
        symptom(flu, moderate_cough).
```

Output:-

```
Goal:  symptom(cold, runny_nose)
Yes
Goal:  symptom(flu, mild_bodyache)
No
```

Practical - 4

Date:

Aim: - **Write a program to display whole List.**

```
domains
        namelist=symbol*
predicates
        writelist(namelist)
clauses
        writelist([]).
        writelist([Head|Tail]):-
                write(Head),nl,writelist(Tail).
```

Output:-

```
Goal: writelist([a,b,c])
a
b
c
Yes
```

Aim: - **Write a program to Append a given List.**

```
domains
        namelist=symbol*
predicates
        append(namelist,namelist,namelist)
clauses
        append([],ListB,ListB).
        append([X|List1],List2,[X|List3]):-
                append(List1,List2,List3).
```

Output:-

```
Goal: append([a,b,c],[d,e],X)
X=["a","b","c","d","e",]
1 Solution
```

Aim: - **Write a program to Reverse a given List.**

```
domains
        namelist=symbol*
predicates
        reverse_list(namelist,namelist)
        reverse(namelist,namelist,namelist)
clauses
        reverse_list(Inputlist,Outputlist):-
                reverse(Inputlist,[],Outputlist).
        reverse([],Inputlist,Inputlist).
        reverse([Head|Tail],List1,List2):-
                reverse(Tail,[Head|List1],List2).
```

Output:-

```
Goal:reverse_list([a,b,c,d],X)
X=["d","c","b","a"]
1 Solution
```

Aim: - **Write a program to find the last element of a given List.**

```
domains
        namelist=symbol*
predicates
        last_element(namelist,symbol)
clauses
        last_element([Head],X):-
                X=Head.
        last_element([_|Tail],X):-
                last_element(Tail,X).
```

Output:-

```
Goal:last_element([a,b,c],X)
X=c
1 Solution
```

Aim: - **Write a program to find the nth element of a given List.**

```
domains
        namelist=symbol*
        NN=integer
predicates
        n_element(namelist,integer,symbol)
clauses
        n_element([Head|_],1,X):-
                X=Head.
        n_element([_|Tail],N,X):-
                NN=N-1,
                n_element(Tail,NN,X).
```

Output:-

```
Goal: n_element([a,b,c],3,X)
X=c
1 Solution
```

Practical - 5

Date:

Aim: - **Logon example without Recursion.**

```
domains
        name, password=symbol
predicates
        getinput(name, password)
        logon
        user(name, password)
clauses
        logon:-
                clearwindow,
                getinput(_,_),
                write("You are now  logged on."),nl.
        logon:-
             write("Sorry, you are not permitted access."),nl,
        getinput(Name, Password):-
                write("Please Enter Your Name:"),
                readln(Name),nl,
                write("Please Enter Password:"),
                readln(Password),nl,
                user(Name, Password).
        user(john, superman).
        user(sue, happy).
        user(bill, bigfoot).
```

Output:-

```
Goal: logon
Enter Your Name: patel
Please Enter Password: pqr
Sorry, you are not permitted to access.
```

Practical - 6

Date:

Aim: - **Logon example with Repeat Predicate.**

One predicate that uses recursion is the repeat predicate. Its format is
repeat.
repeat :-
repeat.

The repeat predicate is useful for forcing a program to generate alternate solutions through backtracking. When the repeat predicate is used in a rule, the predicate always succeeds. If a later premise causes failure of the rule, Prolog will backtrack to the repeat predicate. The repeat predicate always succeeds and terminates the backtracking, causing Prolog to move forward again on the same path.

```
domains
        name, password=symbol
predicates
        getinput (name, password)
        logon
        user(name, password)
        repeat
clauses
        repeat.
        repeat:-
                repeat.
        logon:-
                clearwindow,
                getinput(_,_),
                write("You are now logon."),nl.

        logon:-
                repeat,
                write("Sorry, you are not permitted to access."),nl,
                write("Please try again."),nl,
                getinput (_,_),
                write("You are logged on."),nl.
        getinput(Name, Password):-
                write("Please Enter your name:"),
                readln(Name),nl,
                write("Please Enter Password:"),
                readln(Password),nl,
                user(Name,Password).
        user(john,superman).
        user(sue,happy).
```

```
user(bill,bigfoot).
```

Output:-

```
Goal: logon
Please Enter your name: abc
Please Enter Password: xyz
Sorry, you are not permitted to access.
Please try again.
Please Enter your name: john
Please Enter Password: superman
You are logged on.
Yes
```

Practical - 7

Date:

Aim: - Write a program to solve Tower of Hanoi problem.

Objective: The objective or goal of this problem is to transfer all the 'n' discs from source pole to the destination pole in such a way that we get the same arrangement of discs as before. But this goal must be achieved by sticking to the rules.

Rules and Constraints: - The constraints that must be satisfied while solving the problem are –

1. Only one disc can be moved at a time.
2. Only the top-most disc can be removed
3. The larger disc cannot be placed on top of the smaller disc.

```
def TowerOfHanoi(n , s_pole, d_pole, i_pole):
   if n == 1:
      print("Move disc 1 from pole",s_pole,"to pole",d_pole)
      return
   TowerOfHanoi(n-1, s_pole, i_pole, d_pole)
   print("Move disc",n,"from pole",s_pole,"to pole",d_pole)
   TowerOfHanoi(n-1, i_pole, d_pole, s_pole)

n = 3
TowerOfHanoi(n, 'A', 'C', 'B')
# A, C, B are the name of poles
```

Output:-

```
Move disk 1 from pole A to pole C
Move disk 2 from pole A to pole B
Move disk 1 from pole C to pole B
Move disk 3 from pole A to pole C
Move disk 1 from pole B to pole A
Move disk 2 from pole B to pole C
Move disk 1 from pole A to pole C
```

Practical - 8

Date:

Aim: - Write a program to solve Travelling Salesman Problem.

```
from sys import maxsize
from itertools import permutations
V = 4
def travellingSalesmanProblem(graph, s):
        vertex = []
        for i in range(V):
                if i != s:
                vertex.append(i)
        min_path = maxsize
        next_permutation=permutations(vertex)
        for i in next_permutation:
                current_pathweight = 0
                k = s
                for j in i:
                        current_pathweight += graph[k][j]
                        k = j
                current_pathweight += graph[k][s]
                min_path = min(min_path, current_pathweight)
        return min_path
if __name__ == "__main__":
        graph = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]
        s = 0
        print(travellingSalesmanProblem(graph, s))
```

Output:-

The minimum cost is : 80

Practical - 9

Date:

Aim: - Write a program to solve N Queens problem.

```
class QueenChessBoard:
   def __init__(self, size):  self.size = size
      def get_size(self):   return self.size
   def get_queens_count(self):   return len(self.columns)
   def place_in_next_row(self, column):  self.columns.append(column)
   def remove_in_current_row(self):  return self.columns.pop()
   def is_this_column_safe_in_next_row(self, column):
        row = len(self.columns)
        for queen_column in self.columns:
        if column == queen_column:
            return False
        for queen_row, queen_column in enumerate(self.columns):
        if queen_column - queen_row == column - row:
            return False
        for queen_row, queen_column in enumerate(self.columns):
        if ((self.size - queen_column) - queen_row  == (self.size - column) - row):
            return False
        return True
   def display(self):
      for row in range(self.size):
         for column in range(self.size):
            if column == self.columns[row]:
               print('Q', end=' ')
            else:
               print('.', end=' ')
         print()
   def print_all_solutions_to_n_queen(size):
      board = QueenChessBoard(size)
      number_of_solutions = print_all_solutions_helper(board)
      print('Number of solutions:', number_of_solutions)
   def print_all_solutions_helper(board):
   size = board.get_size()
   if size == board.get_queens_count():
      board.display()
      print()
      return 1
     number_of_solutions = 0
     for column in range(size):
      if board.is_this_column_safe_in_next_row(column):
         board.place_in_next_row(column)
         number_of_solutions += print_all_solutions_helper(board)
        board.remove_in_current_row()
    return number_of_solutions
 n = int(input('Enter n: '))
print_all_solutions_to_n_queen(n)
```

Practical - 10

Date:

Aim: - Write a program to implement DFS.

```
class Graph:
   def __init__(self):
        self.vertices = {}
    def add_vertex(self, key):
       vertex = Vertex(key)
       self.vertices[key] = vertex
    def get_vertex(self, key):
        return self.vertices[key]
    def __contains__(self, key):
      return key in self.vertices
    def add_edge(self, src_key, dest_key, weight=1):
       self.vertices[src_key].add_neighbour(self.vertices[dest_key], weight)
    def does_edge_exist(self, src_key, dest_key):
       return self.vertices[src_key].does_it_point_to(self.vertices[dest_key])
    def __iter__(self):
      return iter(self.vertices.values())
 class Vertex:
    def __init__(self, key):
       self.key = key
       self.points_to = {}
     def get_key(self):
        return self.key
     def add_neighbour(self, dest, weight):
        self.points_to[dest] = weight
     def get_neighbours(self):
       return self.points_to.keys()
     def get_weight(self, dest):
        return self.points_to[dest]
     def does_it_point_to(self, dest):
        return dest in self.points_to
     def display_dfs(v):
       display_dfs_helper(v, set())
     def display_dfs_helper(v, visited):
     visited.add(v)
     print(v.get_key(), end=' ')
     for dest in v.get_neighbours():
       if dest not in visited:
          display_dfs_helper(dest, visited)
 g = Graph()
print('Menu')
print('add vertex <key>')
```

```
print('add edge <src> <dest>')
print('dfs <vertex key>')
print('display')
print('quit')
 while True:
   do = input('What would you like to do? ').split()
    operation = do[0]
   if operation == 'add':
      suboperation = do[1]
      if suboperation == 'vertex':
         key = int(do[2])
         if key not in g:
            g.add_vertex(key)
         else:
            print('Vertex already exists.')
      elif suboperation == 'edge':
         src = int(do[2])
         dest = int(do[3])
         if src not in g:
            print('Vertex {} does not exist.'.format(src))
         elif dest not in g:
            print('Vertex {} does not exist.'.format(dest))
         else:
            if not g.does_edge_exist(src, dest):
               g.add_edge(src, dest)
            else:
               print('Edge already exists.')
    elif operation == 'dfs':
      key = int(do[1])
      print('Depth-first Traversal: ', end='')
      vertex = g.get_vertex(key)
      display_dfs(vertex)
      print()
   elif operation == 'display':
      print('Vertices: ', end='')
      for v in g:
         print(v.get_key(), end=' ')
      print()
       print('Edges: ')
      for v in g:
         for dest in v.get_neighbours():
            w = v.get_weight(dest)
            print('(src={}, dest={}, weight={}) '.format(v.get_key(), dest.get_key(), w))
      print()
    elif operation == 'quit':
      break
```

Practical - 11

Date:

Aim: - Write a program to implement 8-Puzzle problem.

```
import sys
import numpy as np
class Node:
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action
class StackFrontier:
    def __init__(self):
        self.frontier = []
    def add(self, node):
        self.frontier.append(node)
    def contains_state(self, state):
        return any((node.state[0] == state[0]).all() for node in self.frontier)
    def empty(self):
        return len(self.frontier) == 0
    def remove(self):
        if self.empty():
            raise Exception("Empty Frontier")
        else:
            node = self.frontier[-1]
            self.frontier = self.frontier[:-1]
                return node
class QueueFrontier(StackFrontier):
    def remove(self):
        if self.empty():
            raise Exception("Empty Frontier")
        else:
            node = self.frontier[0]
            self.frontier = self.frontier[1:]
              return node
class Puzzle:
    def __init__(self, start, startIndex, goal, goalIndex):
        self.start = [start, startIndex]
        self.goal = [goal, goalIndex]
        self.solution = None
    def neighbors(self, state):
        mat, (row, col) = state
          results = []
        if row > 0:
          mat1 = np.copy(mat)
          mat1[row][col] = mat1[row - 1][col]
          mat1[row - 1][col] = 0
                results.append(('up', [mat1, (row - 1, col)]))
        if col > 0:
```

```
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row][col - 1]
            mat1[row][col - 1] = 0
            results.append(('left', [mat1, (row, col - 1)]))
    if row < 2:
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row + 1][col]
            mat1[row + 1][col] = 0
            results.append(('down', [mat1, (row + 1, col)]))
    if col < 2:
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row][col + 1]
            mat1[row][col + 1] = 0
            results.append(('right', [mat1, (row, col + 1)]))
            return results
    def print(self):
            solution = self.solution if self.solution is not None else None
            print("Start State:\n", self.start[0], "\n")
            print("Goal State:\n",  self.goal[0], "\n")
            print("\nStates Explored: ", self.num_explored, "\n")
            print("Solution:\n ")
    for action, cell in zip(solution[0], solution[1]): print("action: ", action, "\n", cell[0],   "\n")
            print("Goal Reached!!")
    def does_not_contain_state(self, state):
            for st in self.explored:
                    if (st[0] == state[0]).all():
                            return False
            return True
    def solve(self):
            self.num_explored = 0
            start = Node(state=self.start, parent=None, action=None)
            frontier = QueueFrontier()
            frontier.add(start)
            self.explored = []
            while True:
            if frontier.empty():
                    raise Exception("No solution")
                    node = frontier.remove()
                    self.num_explored += 1
            if (node.state[0] == self.goal[0]).all():
                    actions = []
                    cells = []
                    while node.parent is not None:
                            actions.append(node.action)
                            cells.append(node.state)
                            node = node.parent
                            actions.reverse()
                            cells.reverse()
                            self.solution = (actions,  cells)
                            return
```

```
                    self.explored.append(node.state)
                    for action, state in self.neighbors(node.state):
                    if not frontier.contains_state(state) and self.does_not_contain_state(state):
                                child = Node(state=state, parent=node, action=action)
                                frontier.add(child)
start = np.array([[1, 2, 3], [8, 0, 4], [7, 6, 5]])
goal = np.array([[2, 8, 1], [0, 4, 3], [7, 6, 5]])
startIndex = (1, 1)
goalIndex = (1, 0)
p = Puzzle(start, startIndex, goal, goalIndex)
p.solve()
p.print()
```

Practical - 12

Date:

Aim: - Write a program to implement min max algorithm for any game development.

```
class TreeNode:
  def __init__(self, data, left = None, right = None):
    self.val = data
    self.left = left
    self.right = right
class Solution:
  def helper(self, root, h, currentHeight):
    if not root:
      return
    self.helper(root.left, h, currentHeight + 1)
    self.helper(root.right, h, currentHeight + 1)
    if currentHeight < h:
      if currentHeight % 2 == 0:
        if root.left and root.right:
          root.val = max(root.left.val, root.right.val)
        elif root.left:
          root.val = root.left.val
        elif root.right:
          root.val = root.right.val
      else:
        if root.left and root.right:
          root.val = min(root.left.val, root.right.val)
        elif root.left:
          root.val = root.left.val
        elif root.right:
          root.val = root.right.val
  def height(self, root):
    if not root:
      return 0
    return 1 + max(self.height(root.left), self.height(root.right))
  def solve(self, root):
    h = self.height(root)
    self.helper(root, h, 0)
    return root
def print_tree(root):
  if root is not None:
    print_tree(root.left)
    print(root.val, end = ', ')
```

```
    print_tree(root.right)
ob = Solution()
root = TreeNode(0)
root.left = TreeNode(3)
root.right = TreeNode(0)
root.right.left = TreeNode(0)
root.right.right = TreeNode(0)
root.right.left.left = TreeNode(-3)
root.right.right.right = TreeNode(4)
print_tree(ob.solve(root))
```

Output : 3, 3, -3, -3, -3, 4, 4

Practical - 13

Date:

Aim: - Write a program to implement A* algorithm.

```
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = {}
    parents = {}
    g[start_node] = 0
    parents[start_node] = start_node
    while len(open_set) > 0:
        n = None
        for v in open_set:
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v
        if n == stop_node or Graph_nodes[n] == None:
            pass
        else:
            for (m, weight) in get_neighbors(n):
                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                else:
                    if g[m] > g[n] + weight:
                        #update g(m)
                        g[m] = g[n] + weight
                        parents[m] = n
                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)
        if n == None:
            print('Path does not exist!')
            return None
        if n == stop_node:
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]
            path.append(start_node)
            path.reverse()
            print('Path found: {}'.format(path))
```

```
            return path
        open_set.remove(n)
        closed_set.add(n)
    print('Path does not exist!')
    return None
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
def heuristic(n):
    H_dist = { 'A': 11, 'B': 6, 'C': 5, 'D': 7,'E': 3, 'F': 6, 'G': 5, 'H': 3,'I': 1,'J': 0}
    return H_dist[n]
Graph_nodes = {  'A': [('B', 6), ('F', 3)], 'B': [('A', 6), ('C', 3), ('D', 2)],  'C': [('B', 3), ('D', 1), ('E', 5)],
    'D': [('B', 2), ('C', 1), ('E', 8)], 'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)], 'F': [('A', 3), ('G', 1), ('H', 7)],
    'G': [('F', 1), ('I', 3)], 'H': [('F', 7), ('I', 2)], 'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],}
aStarAlgo('A', 'J')
```

Output:

```
Path found: ['A', 'F', 'G', 'I', 'J']
```